VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky

Bakalárska práca

2013 Michal Čičkán

VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zobrazování transparentních materiálů pomocí OpenGL
Correct Rendering of Semi-transparent Materials in OpenGL

2013 Michal Čičkán

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Michal Čičkán**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Zobrazování transparentních materiálů pomocí OpenGL
Correct Rendering of Semi-transparent Materials in OpenGL

Zásady pro vypracování:

Standardní zobrazovací pipeline OpenGL neobsahuje podporu pro korektní vykreslování poloprůhledných materiálů. U reálných modelů a komplexních scén nelze předpokládat vhodné seřazení vykreslovaných ploch. Z tohoto důvodu je nutné řešit zobrazování takovýchto objektů explicitně. Cílem práce je vytvoření detailního popisu algoritmu pro vykreslování poloprůhledných materiálů a jeho následná implementace pomocí moderních funkcí OpenGL.

1. Seznamte se s metodami vykreslování poloprůhledných ploch.
2. Vybranou metodu implementujte v C++ pomocí OpenGL (omezte se na funkce z core profile 4.x).
3. Funkčnost demonstrujte na sadě vhodných modelů.

Seznam doporučené odborné literatury:

[1] Bavoil, L., Myers, K. Order Independent Transparency with Dual Depth Peeling. 2008.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Tomáš Fabián**

Datum zadání: 16.11.2012
Datum odevzdání: 07.05.2013

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prehlásenie študenta

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne.

Uviedol som všetky literárne pramene a publikácie z ktorých som čerpal.

V Ostravedňa 02. 05. 2013

.....
Podpis

Poďakovanie

Rád by som poďakoval vedúcemu bakalárskej práce Ing. Tomášovi Fabiánovi za účinnú pedagogickú a odbornú pomoc a všetky ďalšie odborné rady pri spracovaní tejto bakalárskej práce.

Abstrakt

Zobrazovanie priehľadných plôch je vedecká disciplína, ktorá sa zaoberá korektným zobrazením transparentnosti. Plochý sú definované vrcholmi, ktoré obsahujú informácie o farbe, transparentnosti a pozícii. Na základe daných informácií je určená výsledná farba a transparentnosť jednotlivých objektov.

Táto bakalárska práca sa zaoberá zobrazením priehľadných plôch spôsobom, ktorý nie je závislý na usporiadaní vrcholov objektu. Poskytuje porovnanie medzi jednoduchou implementáciou zabudovanou v OpenGL a implementáciou nezávislou na poradí vrcholov objektu. Praktická časť sa sústredí na implementáciu Dual Depth Peelingu v jazyku c++ používajúcu grafické knižnice OpenGL a jeho výsledku pri aplikovaní na transparentný objekt.

Kľúčové slová: OpenGL, C++, správne zobrazovanie priehľadných plôch, nezávislé na poradí

Abstract

Rendering of transparent surfaces is scientific discipline, which studies correct displaying of transparency. Surfaces are defined with vertices, which contains information about color, transparency and position. Based on this informations is identified result color and transparency of every single object.

This thesis provides solution of displaying transparent surfaces, which is not depended on order of vertices of object. Provides comparison between simple implementation embedded in OpenGL and implementation, which is independent on order of vertices. The practical part is concentrating to implement method Dual Depth Peeling in programming language c++, using graphic libraries OpenGL and his result after application on transparent object.

Key words: OpenGL, C++, correct rendering semi-transparent surfaces, independent order

Zoznam použitých skratiek

RGB	Red Green Blue color scheme
RGBA	Red Green Blue Alpha color scheme
GPU	Graphic Proccessor Unit
OpenGL	Openg Graphic Libraries
GLSL	Graphic Library Shading Language
VAO	Vertex Array Object
VBO	Vertex Buffer Object
GDC	Game Developers Conference

Obsah

1. Úvod

Žijeme v dobe, kedy grafické zobrazenie dát využíva široká škála ľudí. Používa sa takmer v každom odbore od medicíny, strojárstva, až po pokročilý herný priemysel. Technológie grafického zobrazenia sa stále vylepšujú alebo sa vymýšľajú nové. Tento trend smeruje k stále lepšiemu zobrazeniu, ktoré sa miestami približuje k realite. Takáto tendencia kladie čoraz väčší dôraz na výkon počítača, a tak sa neustále vymýšľajú optimalizačné algoritmy, ktoré by znížili nárok na výkon. Pri optimalizácii sa však musí dať pozor, aby sa veľmi neznížila kvalita zobrazenia dát.

Bakalárska práca sa zaoberá vykresľovaním priehľadných plôch, nezávisle od poradia fragmentov. Poskytuje zbežný prehľad a popis ostatných riešení tohto problému, ale jej hlavným cieľom je, aby nás zoznámila s problematikou a implementáciou Dual depth peelingu. Taktiež ukáže výhody oproti riešeniam, ktoré sú závislé na usporiadaní fragmentov v jednom smere.

Konečným výsledkom bakalárskej práce bude implementácia riešenia Dual depth peelingu v grafických knižniciach OpenGL. V tejto implementácii sú zahrnuté aj iné riešenia, ktoré sa budú v závere práce porovnávať.

1.1 Grafická knižnica OpenGL

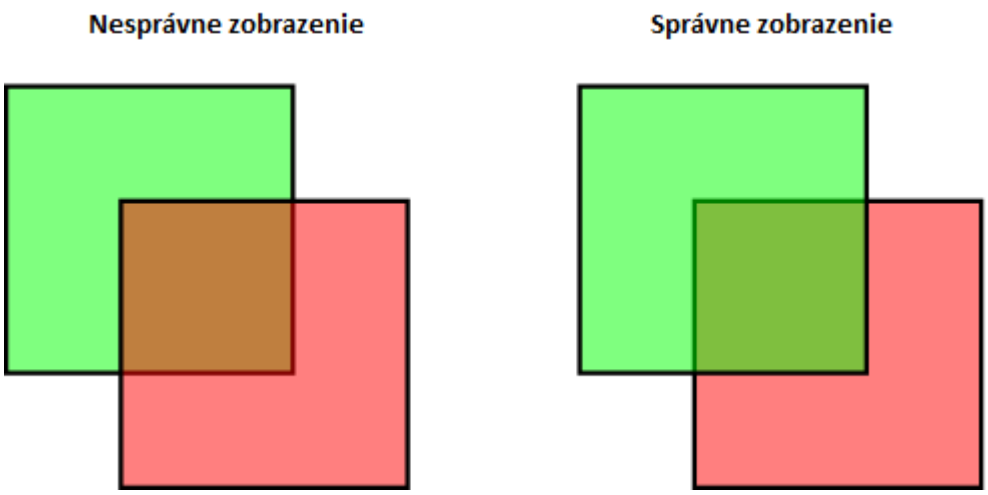
Toto softvérové rozhranie ku grafickému zariadeniu sa používa na viacerých platformách, čo je jeho výhoda oproti iným rozhraniam, pracujúcich len na jednej platforme. Slúži na vykresľovanie 3D objektov, ktoré sú zobrazované užívateľovi k lepšiemu porozumeniu dát, ktoré by mu inak nič nepovedali. Vybral som si ho pre jeho veľkú efektivitu, vzhľadom na to, že pristupuje priamo k hardvéru a nemá medzi sebou žiadnu ďalšiu softvérovú vrstvu. Táto knižnica je napríklad využívaná v mobilných operačných systémoch Android a iOS, kde sa stretla s veľkým nadšením vývojárov. Od jeho prvého vydania uplynulo už 20 rokov a za tú dobu sa posunulo závratne vpred. Vďaka tomu je veľmi rozšírené.

2. Čo je alfa kanál

Pri rozhodnutí o pridaní priehľadnosti objektu je riešenie jednoduché. Pridáme k farebnej schéme RGB, ktorá využíva tri farby na miešanie výslednej farby pixelu červenú, modrú, zelenú, kanál alfa, ktorý nám určuje stupeň priehľadnosti objektu, kde je maximálna priehľadnosť rovná hodnote 0 a maximálna nepriehľadnosť rovná hodnote 1. Týmto postupom nám vznikne farebná schéma RGBA, s ktorou môžeme využívať alfa kanál. Táto farebná schéma je použitá pre každý jeden pixel, ktorý sa zobrazuje na zobrazovacom zariadení. V OpenGL sa pre umožnenie využívania miešania farieb na základe alfa kanálu volá funkciou glEnable(GL_BLEND).

2.1 Problém

Problém nastáva, keď pri zobrazovaní priehľadnosti sú fragmenty nezoradené, kedy technika zobrazovania priehľadných objektov závisí od zoradenia fragmentov. Môže nastať, že vznikne iná výsledná farba, aká by mala vzniknúť.



Obrázok 1: Zobrazenie správnej a nesprávnej priehľadnosti pri nezoradených fragmentoch

Pri zmiešaní dvoch farieb vznikne nová farba, ktorá je výsledkom spočítania týchto dvoch farieb a ich hodnôt priesvitnosti.

2.2 Riešenie problému

Zvyčajne sa problém rieši zoradením všetkých priehľadných trojuholníkov, kde hĺbkový zásobník, ktorý slúži na uchovávanie vzdialenosti fragmentu od projekčnej roviny negovanou

hodnotou, vyfiltruje trojuholníky, ktoré sú skryté a nezobrazujú sa. Následne na to sa zoradia trojuholníky od najvzdialenejšieho po najbližší, od projekčnej roviny. Pri zoraďovaní trojuholníkov nastáva problém s limitovaním množstva naplnenia hĺbkového zásobníku. Každý fragment môže byť zapísaný do hĺbkového zásobníku až 20 krát, čo je náročné na pamäť. Tento úkon sa vykonáva 4-krát pre každý pixel, pokiaľ sa nevyužije optimalizácia, čo je tiež náročné na výkon. Takisto triedenie všetkých priehľadných trojuholníkov trvá dlhú dobu.

Ďalší problém je, keď je potrebné zmeniť textúru alebo shade, vznikne veľmi veľká záťaž na výkon. Najlepšie riešenie je, keď sa obmedzí počet priehľadných polygónov. Používa sa tá istá textúra a shader pre všetky polygóny. Ak majú vyzerať odlišne, je lepšie použiť textúru. Táto technika je nevyhovujúca, pokiaľ má byť zobrazenie kvalitné.

3. Metódy zobrazovania priehľadných plôch

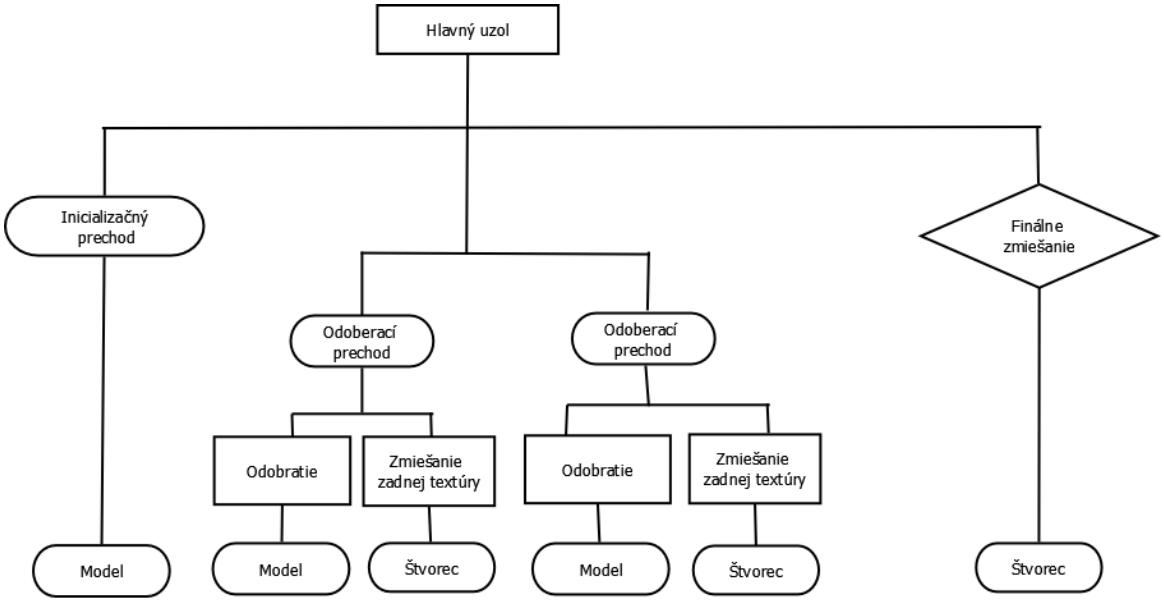
Táto kapitola sa zaoberá druhmi riešenia problematiky zobrazovania priehľadných plôch. Poskytuje prehľad a spôsob implementácie rôznych druhov algoritmov.

3.1 Dual Depth Peeling

Dual depth peeling bol vyvinutý Kevinom Mayersom, Louisom Bavoilom a Mehmetom Cem Cebenoyanom. Je to technika riešenia zobrazovania priehľadných plôch, ktorá nezávisí od zoradenia fragmentov smerom spredu dozadu a zozadu dopredu. Týmto odpadá nutnosť zoradenia fragmentov, čo zlepšuje rýchlosť vykreslenia. Vykresľuje priehľadné objekty viackrát, kde „ureže“ dve vrstvy fragmentov po každom raze. Využíva sa tu minimálny a maximálny hĺbkový zásobník, ktorý súčasne berie jednu vrstvu spredu a jednu zozadu. To znamená, že pri N-vrstvách je potrebný N/2+1 krokov, čo je o polovicu menej krokov ako je potrebné pri jeho predchodcovi depth peeling. Takáto náročnosť sa nazýva hĺbková zložitosť.

3.1.1 Algoritmus

Toto riešenie zobrazovania priehľadných plôch začne orezávaním vrstiev z hĺbkového zásobníku v oboch smeroch, a to smerom spredu dozadu a odzadu dopredu, po každom prebehnutí hĺbkového zásobníku. Postupne získava najbližšiu vrstvu fragmentov, uloží jej farbu, a potom hľadá ďalšiu najbližšiu vrstvu. Potom sa navzájom zmiešajú a takto proces pokračuje, až kým príde nakoniec. Pri podpore viacerých prípravných vyrovnávacích pamätí je možné postupovať oboma smermi. Táto vyrovnávacia pamäť slúži pre ukladanie rôznych farieb do osobitných zásobníkov, kde sa vytvára toľko zásobníkov, koľko je farieb.



Obrázok 2: Priebeh algoritmu popísaný diagramom

3.1.2 Inicializácia

Na začiatku procesu sa inicializuje hĺbková textúra na hodnoty minimálnej a maximálnej hĺbky na -1. Neskôr sa hĺbková textúra nastaví na negovanú minimálnu hĺbku a na maximálnu hĺbku objektu pre každý fragment(-gl_FragCoord.z,gl_FragCoord.z) v 32 bitovej hĺbkovej textúre a použije sa MAX_BLEND, ktorý uchováva tieto dve hodnoty, a to najväčšiu a znegovanú najmenšiu hodnotu hĺbky, zo všetkých hĺbkových hodnôt fragmentov. Hodnota float2(-depth, depth) sa zapíše do hĺbkovej textúry. Staršie grafické karty nepodporovali 32 bitové miešanie dátového typu float. Táto vymoženosť sa začala používať až od grafickej karty GeForce 8. Fragment shader zapisuje do dvoch farebných textúr a jednej hĺbkovej. Toto zapisovanie je umožnené a zabudované priamo v OpenGL používaním zásobníkov frame objektov. Prvá textúra(RT0) sa používa pre uloženie minimálnej a maximálnej hĺbkový zásobník, Druhá textúra(RT1) sa používa pre zhromažďovanie predných fragmentov a tretia textúra(RT3) pre uchovávanie dočasnej zadnej vrstvy fragmentov. Pre RT0 sa používa formát textúry RG32F, pre RT1 RGBA8 a pre RT2 taktiež formát RGBA8. Tento formát textúry pre RT1 a RT2 sa bude používať preto, lebo potrebujeme uchovávať hodnotu alfa pre každú textúru. Pri inicializácii sa neureže z hĺbkového zásobníku ani jedna vrstva.

3.1.3 Odoberanie vrstiev

Základná myšlienka tohto riešenia, je po každej odobranej vrstve získať najbližšiu a najvzdialenejšiu vrstvu fragmentov, ktoré ešte neboli spracované. Po každom kroku treba aktualizovať hĺbkovú textúru a aktualizovať farby fragmentov umiestnené v prednej a zadnej časti hĺbkového zásobníku. Na takéto operácie sa využíva shader, ktorý počíta efekty pri vykresľovaní na GPU, čo má za následok rýchlejšie počítanie výsledného obrazu (viď kapitola 4).

Aktualizácia hĺbkovej textúry sa vykonáva takisto ako pri inicializácii, akurát ignorujeme vrstvy fragmentov, ktoré už boli spracované. Podobne ako pri inicializácii, aj tu sa používa MAX_BLEND[1].

Pre spočítanie výslednej farby smerom zozadu dopredu do zmiešanej textúry sa využíva nasledovný vzorec:

$$Cdst = Cdst + Csrc*Asrc*Adst$$

Rovnica 1: Vzorec pre výpočet miešania zozadu dopredu

Takže výslednú farbu získame pripočítaním farby z cieľového zásobníku(dst_color) k súčinu farby(src_color) a alfy(src_alpha) zo zdrojového kanálu alfa z cieľového zásobníku(dst_alpha). Pre výpočet prednej textúry sú dva spôsoby, ako je možné problém riešiť. Najbežnejší je spôsob, popísaný v rovnici 1.

$$Cdst= Csrc*Asrc + (1-Asrc)*Cdst$$

Rovnica 2: Alternatívny vzorec pre výpočet farieb prednej textúry

Tu je vidieť, že tento vzorec nevyužíva cieľovú alfu, preto tento vzorec nemôžeme použiť a je potrebné túto rovnicu upraviť.

$$C1' = A1\ C1 + (1-A1)\ C0$$

$$C2' = A2\ C2 + (1-A2)\ C1'$$

$$C2' = A2\ C2 + (1-A2)\ (A1\ C1 + (1-A1)\ C0)$$

$$C2' = A2\ C2 + (1- A2)\ A1\ C1 + (1-A2)\ (1-A1)\ C0$$

Rovnica 3: Vzťah pre výpočet fragmentov zozadu dopredu

Z tohto vzťahu je zrejmé, že fragmenty zozadu dopredu môžu byť zmiešané nasledujúcou osobitnou rovnicou.

$$Cdst = Adst\ (Asrc\ Csrc) + Cdst$$

$$Adst = (1-Asrc)\ Adst$$

Rovnica 4: Vzorec pre výpočet zmiešavania fragmentov zozadu dopredu

Týmto dostaneme výslednú prednú farbu. Pri tejto rovnici je nastavená hodnota cieľovej alfy(*Adst*) na hodnotu 1.Nie je tu možné použiť shader, pretože GLSL, ktorý je jazyk pre implementovanie shaderov v OpenGL, povoľuje len jeden shader na jednu funkciu pre miešanie. A tu je potrebná jedna funkcia pre hĺbkovú textúru a ďalšia funkcia pre prednú farbu. Preto je potrebné použiť o jeden shader naviac. Pri smere spredu dozadu je možné použiť MAX_BLEND, nakoľko farba spredu dozadu sa môže len zvyšovať, ako bolo ukázané na rovnici číslo 4, pre výpočet farby.

3.1.4 Prepínanie textúr

Po urezaní všetkých vrstiev vznikne zmiešaná predná a zadná textúra. Je potrebné, spojiť ich do jednej textúry a zobraziť na druhý štvorec, zarovnaný na obrazovke. Pri tomto úkone môžu nastať hazardy čítania – zápisu. Preto je potrebné vytvoriť rôzne textúry pre čítanie a pre zapisovanie farieb fragmentov. Týmto vzniknú dve hĺbkové, dve predné a dve dočasné zadné textúry. Po každom kroku sa zmenia cieľový a zdrojový zásobník pre vykreslenie. Pri spôsobe dynamického počtu vrstiev je otázka, ktorý zásobník obsahuje finálnu textúru, či zdrojový alebo cieľový zásobník. Musí sa nastaviť zobrazovacie pre každú kameru pri inicializácii. Vytvorí sa finálny krok pre prípad, že je počet urezaných vrstiev párny a jeden pre prípad, že počet urezaných vrstiev bude nepárny. Týmto krokom zamedzíme problému, v ktorom zásobníku sa bude finálny priechod nachádzať. Prepnutie na správny finálny priechod sa zabezpečí počas výpočtu, podľa toho, koľko vrstiev bolo rezaných. Tento spôsob nie je ideálny pre vykresľovanie v reálnom čase, ale skôr pre scény, ktoré sa rendrujú do výsledného zobrazenia, ako napríklad animované filmy.

3.2 Metóda váženého priemeru

Tento spôsob riešenia bol vyvinutý v roku 2007 pánom Meshkinom. Predstavil ho na konferencii GDC. Táto technika používa stratégiu rozšírenia rovnice pre miešanie alfa kanálu a ignorovanie termov závislých na poradí. Spôsob uvedenej metódy je veľmi rýchly a potrebuje len jeden priechod pre výpočet. Sú tu generované dobré výsledky, ktoré majú nízku hodnotu alfa. Z tejto techniky sú výstupy buď veľmi tmavé alebo veľmi svetlé, podľa toho, ktorý term sa nezapočítava do výsledku, okrem prípadu, kedy je alfa menšia ako 30 percent. Spoločnosť NVIDIA túto techniku zobrazenia skvalitnila. Pri ich spôsobe implementácie sa berú do úvahy všetky termy, takže výsledky sú lepšie aj pri vyššom alfa kanáli.

Pri uniformnom alfa kanáli je možné veľmi jednoducho zmiešať farby a nebyť závislý na poradí fragmentov. V prípade, že farby a alfa kanál nie sú uniformné, tak sa nahradia viaceré

farby uniformnou farbou pre každý pixel a to tak, že sa vypočíta vážený priemer farieb u každého pixelu. V prípade, že nie sú uniformné ani alfa kanály pixelu, použije sa taktiež metóda váženého priemeru podľa toho, aké majú farby hodnoty viditeľnosti pre každý pixel. Priemerná farba pre každý pixel je generovaná zobrazením transparentnej geometrie do akumulačného zásobníka, ktorý reprezentuje šestnásťbitová textúra s pohyblivou čiarkou. Výsledkom je akumulovaná tridsaťdvabitová farba (RGBA) a počet fragmentov na každý pixel, ktoré sa nazývajú hĺbková komplexnosť[2]. Po naplnení a vykreslení do akumulačného zásobníku sa textúra posiela na ďalšie spracovanie, kde sa vykoná výpočet váženého priemeru podľa alfa kanálu, ktorého výsledkom je vážený priemer farby.

$$C = \sum[(RGB)\ A] \ / \sum A$$

Rovnica 5: Vzorec pre výpočet váženého priemeru farby pre každý fragment

Písmeno "*C*" predstavuje výslednú farbu, ktorá je rovná súčtu všetkých farieb, vynásobených váženým priemerom alfa kanálov a vydelený súčtom všetkých alfa kanálov. Výpočet váženého priemeru alfa kanálu je nasledovný:

$$A = \sum A \ / \ n$$

Rovnica 6: Vzorec pre výpočet váženého priemeru alfa kanálu

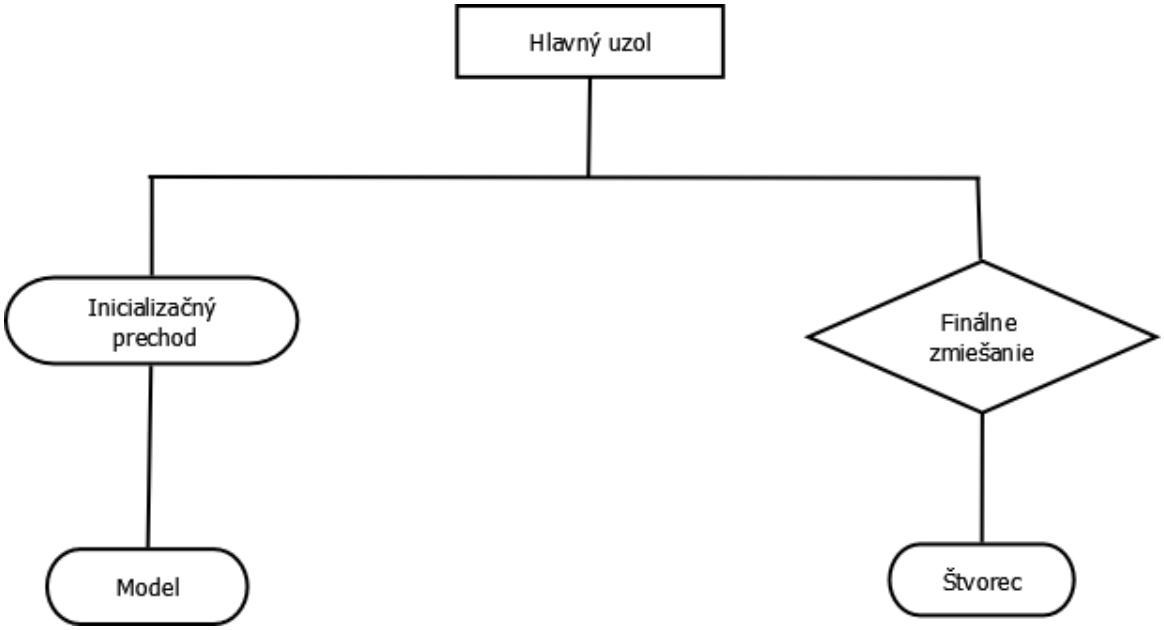
V tejto rovnici, sa počíta so súčtom hodnôt alfa kanálov, a následným vydelením hĺbkovou komplexnosťou (n). S touto rovnicou vypočítame vážený priemer farieb. Pre výslednú farbu použijeme nasledovnú rovnicu.

$$Cdst = C\ A \sum[(1-A)\text{^}k, k = 0..n-1] + Cbg\ (1-A)\text{^}n$$

$$Cdst = C\ A\ (1-(1-A)\text{^}n)/A + Cbg\ (1-A)\text{^}n$$

Rovnica 7: Vzorec pre výslednú farbu pre každý fragment

Konečná farba (Cdst) je súčin všetkých položiek. Použitá rovnica už ráta s farbou pozadia (Cbg), ktorá je upravená podľa toho, koľko vrstiev farieb sa na nej nachádza. Tento spôsob je dobrý pre rendrovanie v reálnom čase.



Obrázok 3: Priebeh algoritmu popísaný diagramom

3.3 Metóda váženého súčtu

Metóda, ktorú taktiež vyvinul Meshkin. Je veľmi podobná ako metóda váženého priemeru, akurát pri tomto spôsobe sa neuchováva komplexnosť hĺbky. Taktiež používa akumulačný zásobník a pre výpočet nevyužíva vážený priemer, ale vážený súčet. Vzorec pre výpočet výslednej farby je nasledovný:

$$Cdst = \sum[Asrc\ Csrc] + Cbg\ (1 - \sum Asrc)$$

Rovnica 8: Vzorec pre výpočet výslednej farby

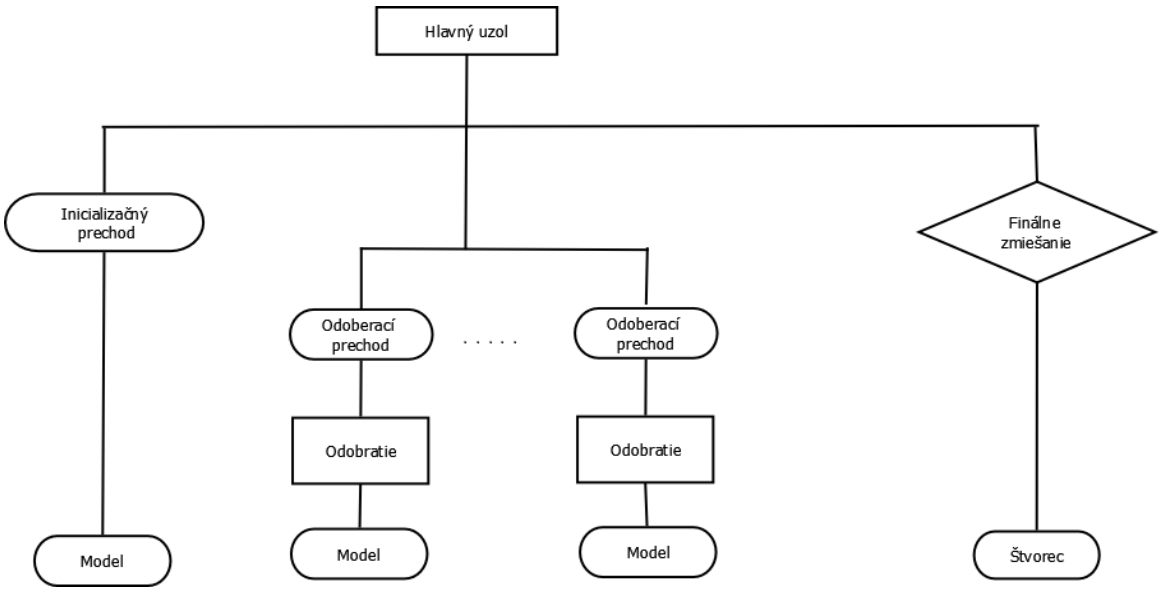
Výsledná farba (Cdst) sa vypočíta ako súčet násobku zdrojovej alfy (Asrc) a zdrojovej farby (Csrc), ku ktorej sa pričíta farba pozadia (Cbg), ktorá je vynásobená súčtom alfa kanálov zdrojovej farby.

Pri tejto metóde vznikajú buď veľmi tmavé alebo veľmi žiarivé výsledky. Metóda je veľmi rýchla a potrebuje len jeden priechod pre výpočet, čo je veľmi výhodné pri zobrazovaní scény v reálnom čase

3.4 Depth Peeling

Metódu vyvinul Cass Everitt zo spoločnosti NVIDIA. Pri uvedenom spôsobe sa odoberá jedna vrstva za druhou, a to smerom odpredu dozadu. Počet prechodov je závislý od toho, aká je hĺbková komplexnosť. Pseudokód je nasledovný[2]:

```
for ( i = 0; i < num_passes; i++ )
{
      clear color buffer
      depth unit 0:
            if(i == 0) { disable depth test }
            else { enable depth test }
            bind depth buffer (i % 2)
            disable depth writes /* read-only depth test */
            set depth func to GREATER
      depth unit 1:
            bind depth buffer ((i+1) % 2)
            clear depth buffer
            enable depth writes;
            enable depth test;
            set depth func to LESS
      render scene
      save color buffer RGBA as layer i
}
```



Obrázok 4: Diagram priebehu algoritmu

Prvý krok získa najbližšiu vrstvu pomocou hĺbkového testu, zabudovanom v OpenGL. Po predchádzajúcom kroku získa ďalšiu najbližšiu vrstvu na základe pixelov. Neskôr použije dva

hĺbkové zásobníky, ktoré porovnávajú súčasnú a predošlú vrstvu. Pri každom úspešnom získaní vrstvy, sa následne táto vrstva odstráni a pomocou hĺbkového testu sa získa ďalšia vrstva. Používajú sa dve textúry, aby nenastal dátovy hazard, ktorý by mohol spôsobiť modifikovanie alebo zmazanie textúry. Nevýhodou tohto riešenia je jeho náročnosť na čas a výpočet. Naopak, výhodou je, že ako výstup je realistické zobrazenie transparentných materiálov. Metóda sa už nepoužíva od príchodu Dual Depth Peelingu, ktorý výpočet urýchlil.

4. Rozširujúci jazyk GLSL

V tejto kapitole bude krátky úvod do jazyka GLSL, ktorý sa bude využívať v implementácii. Tento jazyk je pomerne nový a veľmi efektívny. Táto kapitola sa bude taktiež podrobnejšie venovať dvom typom GLSL, ktoré sa budú využívať pri implementácii, a to fragment shadru a vertex shaderu.

Rozširujúci jazyk pre grafickú knižnicu OpenGL na písanie shaderov je OpenGL Shading Language (ďalej len GLSL). Používa sa pre vytvorenie shaderov, pre každé programovateľné grafické procesory podporujúce OpenGL 2.0 a vyššie. Jazyk vychádza zo syntaxe programovacieho jazyka C. Bol vytvorený združením OpenGL ARB. Táto výhoda priameho výpočtu na grafickom procesore prišla až od OpenGL verzie 2.0, kde sa začali využívať programovateľné reťazce. V nižších verziách boli fixné pipelany, kde nebolo možné modifikovať OpenGL reťazce. Jazyk poskytuje všetky operátory jazyka C. Má vlastný špeciálny operátor swizzle. Swizzle označuje v počítačovej grafike prepočítanie jednotlivých zložiek vektoru. Napríklad pri vektore A = {1,2,3,4}, kde jeho zložky sú v pevnom poradí x, y, z, w, je možné ho prepočítať na vektor B, ktorý má vzorec B = A*w*w*x*y, z čoho vznikne vektor B = {4, 4, 1, 2}. Poskytuje všetky skalárne dátové typy ako float, integer, double. Má taktiež vlastné vektorové typy. Tieto dátové typy sa skladajú z prefixu “vec” a sufixu, ktorý sa skladá z čísla, značiaceho počet prvkov

```
vec3 color = vec3 {1.0, 2.0, 3.0};
```

Má maticový dátový typ, ktorý má podobné zloženie ako vektorový. Pri štvorcových maticiach sa používa ako sufix len jedno číslo. Pri neštvorcových maticiach sa používa sufix typu 3x4.

```
mat3x4 matica = mat3x4 {{1.0,2.0,3.0,4.0},{1.0,2.0,3.0,4.0},
{1.0,2.0,3.0,4.0}, {1.0,2.0,3.0,4.0}};
```

GLSL ponúka transparentné dátové typy. Sú chápané ako obslužný program k objektom. Prístup k nim prebieha cez sadu vstavaných funkcií. Čítanie alebo zapisovanie do takéhoto dátového typu nie je možné. Pri deklarácii dátového typu vznikajú dva obslužné programy. Vnútorný objekt a objekt, ku ktorému dátový typ pristupuje. Dátové typy tohto druhu, ktoré obsluhujú dvoj alebo trojrozmerné obrázky sú napr. “image1D”, “image2D”. Dátové typy obsluhujúce dvoj alebo trojrozmerné textúry sa nazývajú samplery. V GLSL sa zapisujú “sampler2D”, podľa toho, koľko rozmerná textúra sa definuje.

V OpenGL sa tento jazyk volá funkciou glCreateShader, kde sa vyberá, ktorý shader sa bude vytvárať.

```
GLuint shaderHandle = glCreateShader(GL_VERTEX_SHADER);
glCompileShader(shaderHandle);
```

Funkcia vytvára glCreateShader, ten vytvára shader objekt typu vertex shader, ktorý sa následne preloží pre OpenGL, aby s ním mohol operovať. Ďalšie nutné kroky pre používanie shaderu sú vytvorenie programu pre zostavenie shaderov, pripojenie preložených shaderových objektov a proces linkovania programu, ktorý mapuje premenné hlavného programu a premenných GLSL shaderov.

```
GLuint programHandle = glCreateProgram();
glAttachShader(programHandle, vertexShader);
glLinkProgram(programHandle);
```

Takýmto spôsobom sa vytvárajú všetky dostupné shadery v OpenGL.

GLSL má štyri shadery. Fragment shader, vertex shader, geometry shader a tessellation shader, ktorý začal od verzie jazyka GLSL 4.0 používať na výpočet GPU miesto CPU, čo urýchluje výpočet .

4.1 Vertex shader

Vertex shader sa používa na modifikovanie vrcholov objektov a jej pridružených dát. Spracováva jeden vrchol, žiadny iný v rovnakom čase, preto nevyžaduje znalosť viacerých vrcholov v tom istom čase. Pri spracovaní, shader nevie ku akému objektu vertex patrí. Spracováva ho nezávisle na všetkých ostatných vertexoch. Pred tým, než vrchol spracuje, pozrie sa do dočasnej pamäte vrcholov, a ak sa tam nachádza, tak ho ďalej nespracováva. Tento úkon pomáha k zvýšeniu výkonu. Vertex shader vyžaduje priamu podporu grafickej karty pre používanie.

4.2 Fragment shader

Fragment shader sa používa na spracovanie jednotlivých fragmentov. Fragment vzniká pri rasterizácii, kde sa jednotlivé vrcholy menia na pixely. Niekedy sa používa názov pixel shader, čo odpovedá dátam, ktoré spracováva. Fragmenty nosia hodnoty, ako je hĺbka, pozícia a ostatné hodnoty, ktoré je možné poslať spolu s informáciami o fragmente. Vypočítava hodnoty farieb pixelov a ich ďalších atribútov. Často je používaný napríklad pri bump mapingu, čo je technika na zobrazovanie výstupkov a iných nerovností v textúre, aby sa predišlo zbytočnému veľkému počtu polygónov. Tým, že spracováva jednotlivé body obrazovky, je možné v ňom naimplementovať rôzne efekty, ako je tieň, realistické svetlo a iné efekty pri zobrazovaní scény. Fragment shader dokáže modifikovať hĺbku každého fragmentu v hĺbkovom zásobníku, a taktiež spracovať viac

než jednu farbu, ak je aktívny viac než jeden objekt na zobrazenie. Pri samotnom fragment shaderi nie je možné robiť rozsiahle efekty, pretože pozná len fragment, na ktorom pracuje. Pri viacerých fragment shaderoch je realizovateľné scénu riešiť komplexnejšie, pretože spolu poznajú všetky fragmenty, a tým celú scénu. Je to jediný druh shaderu, ktorý môže spracovávať aj po zobrazení obrázku na obrazovku alebo filtrovať video po rasterizovaní. Vertex shader pre svoj výpočet potrebuje 3D objekt. V OpenGL sa používa konštanta pre názov shaderu GL_FRAGMENT_SHADER. Vďaka týmto vlastnostiam sa bude odohrávať odoberanie vrstiev a ich miešanie.

5. Implementácia

Pre implementáciu som vybral Dual depth peeling vďaka jeho dobrému zobrazovaniu priehľadných scén, narozdiel od ostatných metód a pre jeho o polovicu znížený počet priechodov oproti Depth peelingu. V tejto kapitole sa budem venovať kompletnej implementácii Dual depth peelingu a porovnaniu jeho výsledkov so zabudovanou metódou v OpenGL.

5.1 Incializácia textúr

Vykresľovaním do textúry a nie na obrazovku, je potrebné inicializovať zásobník fragmentov, ktorý nám umožní takzvané vykresľovanie mimo obrazovku:

```
glGenFramebuffers(1, &dualDepthFBOID);
```

Príkaz vygeneruje jedinečný identifikátor pre zásobník fragmentov, ktorý sa potom použije vždy, keď je zásobník potrebný.

Pre spracovanie potrebujeme šesť textúr:

```
glGenTextures (2,texID);
glGenTextures (2,backTexID);
glGenTextures (2,depthTexID);
```

Tieto príkazy vygenerujú identifikátory pre 6 textúr. Premenná „texID“ bude uchovávať dve textúry pre rezanie vrstiev odpredu dozadu. Premenná „backTexID“ bude uchovávať dočasné textúry, ktoré budú slúžiť na uchovávanie zadnej vrstvy pre jeho ďalšie spracovanie, pretože v GLSL nie je možné zmiešať dve textúry odlišnými funkciami. Pre uchovávanie zadnej vrstvy bude slúžiť textúra „depthTexID“. V každej textúre sú dve verzie, kvôli tomu, aby nenastal dátový hazard. To znamená, aby sa neprepísala pôvodná textúra novou, čo by mohlo spôsobiť nesprávne zmiešanie farieb.

Pre používanie textúr je potrebné ich inicializovať a zistiť, aké budú mať vlastnosti. Pre zadnú textúru bude nastavenie nasledovné:

```
glBindTexture(GL_TEXTURE_RECTANGLE, depthTexID[i]);
glTexParameteri(GL_TEXTURE_RECTANGLE , GL_TEXTURE_MAG_FILTER,
GL_NEAREST);
glTexParameteri(GL_TEXTURE_RECTANGLE , GL_TEXTURE_MIN_FILTER,
GL_NEAREST);
glTexParameteri(GL_TEXTURE_RECTANGLE , GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE);
```

```
glTexParameteri(GL_TEXTURE_RECTANGLE , GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE);
glTexImage2D(GL_TEXTURE_RECTANGLE , 0, GL_RG32F, WIDTH, HEIGHT,
0, GL_RGB, GL_FLOAT, NULL);
```

Všetky textúry budú vo formáte "GL_TEXTURE_RECTANGLE", aby bola možnosť pracovať s koordinátami, ako s dátovým typom s pohyblivou desatinnou čiarkou, čo iné textúry neumožňujú. Týmto príkazom sa taktiež nastavuje, ktorá textúra sa bude nastavovať. Pre "GL_TEXTURE_MAG_FILTER", ktorý určuje, či má byť textúra zväčšená, postačí "GL_NEAREST", vracia najbližší fragment. Taktiež pre nastavenie textúry, ktorá má byť zmenšená. Pri funkciách "GL_TEXTURE_WRAP_S" a "GL_TEXTURE_WRAP_T" sa nastaví "GL_CLAMP_TO_EDGE", aby v prípade, keď sa textúra nenastaví na celú obrazovku, sa neopakovala, čo je prednastavené v OpenGL. Bude uchovávať len minimálnu a maximálnu hĺbku, preto stačia dva komponenty. Postačuje formát farby RGB, aj keď v tomto prípade bude modrá zložka vždy nastavená na hodnotu nula.

Pre prednú a zadnú dočasnú textúru sú nastavenia také isté. Pri nastavení farebných parametrov textúry je farebný komponent na "RGBA", a taktiež farebná schéma bude nastavená na formát "RGBA"

```
glTexImage2D(GL_TEXTURE_RECTANGLE , 0, GL_RGBA, WIDTH, HEIGHT, 0,
GL_RGBA, GL_FLOAT, NULL);
```

Textúra, ktorá bude obstarávať miešanie alfa kanálu, bude mať nasledovné nastavenie:

```
glGenTextures(1, &colorBlenderTexID);
glBindTexture(GL_TEXTURE_RECTANGLE, colorBlenderTexID);
glTexParameteri(GL_TEXTURE_RECTANGLE, GL_TEXTURE_WRAP_S,
GL_CLAMP);
glTexParameteri(GL_TEXTURE_RECTANGLE, GL_TEXTURE_WRAP_T,
GL_CLAMP);
glTexParameteri(GL_TEXTURE_RECTANGLE, GL_TEXTURE_MIN_FILTER,
GL_NEAREST);
glTexParameteri(GL_TEXTURE_RECTANGLE, GL_TEXTURE_MAG_FILTER,
GL_NEAREST);
glTexImage2D(GL_TEXTURE_RECTANGLE, 0, GL_RGBA, WIDTH, HEIGHT, 0,
GL_RGBA, GL_FLOAT, 0);
```

Táto textúra bude obdĺžniková, ktorá bude mať komponentu nastavenú na “GL_RGBA” a formát farby “GL_RGBA”.

Pre priradenie textúr do cieľov, ktoré sa budú používať zobrazovaním výsledkov zásobníka fragmentov, potrebujeme naviazať textúry na tieto zobrazovacie ciele:

```
GLenum attachID[2]={GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT3};
glBindFramebuffer(GL_FRAMEBUFFER, dualDepthFBOID);
for(int i=0;i<2;i++) {
      glFramebufferTexture2D(GL_FRAMEBUFFER, attachID[i],
      GL_TEXTURE_RECTANGLE, depthTexID[i], 0);
      glFramebufferTexture2D(GL_FRAMEBUFFER, attachID[i]+1,
      GL_TEXTURE_RECTANGLE, texID[i], 0);
      glFramebufferTexture2D(GL_FRAMEBUFFER, attachID[i]+2,
      GL_TEXTURE_RECTANGLE, backTexID[i], 0);
}
```

Tento súbor príkazov zaistí, že vždy, keď sa zavolá funkcia “glDrawBuffers” s enumerátormi “GL_COLOR_ATTACHMENT_N”, kde miesto “N” priradíme číslo, tak sa výsledok zapíše do textúry, ktorá je naviazaná na tento enumerátor.

Pre alfa zmiešavanie použijeme “GL_COLOR_ATTACHMENT_6”.

```
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT6,
GL_TEXTURE_RECTANGLE, colorBlenderTexID, 0);
```

5.2 Incializácia objektov

Pre vykresľovanie objektov a posielanie údajov pre spracovanie v GLSL, je potrebné inicializovať VAO, ktoré slúži ako zásobník objektov, ktoré sú definované vlastnosťami a uložené vo VBO. Na naplnenie VAO je potrebný tento blok príkazov.

```
glBindVertexArray(quadVAOID);
glBindBuffer (GL_ARRAY_BUFFER, quadVBOID);
glBufferData (GL_ARRAY_BUFFER, sizeof(quadVerts), &quadVerts[0],
GL_STATIC_DRAW);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE,0,0);
glBindBuffer (GL_ELEMENT_ARRAY_BUFFER, quadIndicesID);
glBufferData (GL_ELEMENT_ARRAY_BUFFER, sizeof(quadIndices),
&quadIndices[0], GL_STATIC_DRAW);
```

Po prvom príkaze sa nastaví obsah VAO, ktorý uloží VBO objekt, a po následnom zavolaní vykreslí obsah. Zásobník objektu sa naplní vrcholmi obsahujúcimi premennú "quadVerts" a indexmi, ktoré určujú, aké vrcholy sa majú spájať.

5.3 Algoritmus vykreslenia

HĺbkovýHĺbkový test v klasickom OpenGL vráti len najbližší fragment. Avšak, sú fragmenty vzdialenejšie, a tie OpenGL neumožní spracovávať. Preto je potrebné test vypnúť.

```
glDisable(GL_DEPTH_TEST);
```

Aby bola činná funkcia pre transparentnosť, je ju potrebné v OpenGL aktivovať. To je možné za pomoci nasledujúcej funkcie:

```
glEnable(GL_BLEND);
```

Pre vykresľovanie do zásobníku fragmentov mimo obrazovku, potrebujeme nastaviť, aby bol aktívny.

```
glBindFramebuffer(GL_FRAMEBUFFER, dualDepthFBOID);
```

Týmto zaistíme, že všetko, čo sa bude vykresľovať, smeruje do tohto zásobníku fragmentov.
Pre vykresľovanie potrebujeme tieto textúry nastaviť na nulové hodnoty a nulový alfa kanál, aby sa pri miešaní nerátalo s farbou pozadia.

```
glDrawBuffer(drawBuffers[0]);
glClearColor(-MAX_DEPTH, -MAX_DEPTH, 0, 0);
glClear(GL_COLOR_BUFFER_BIT);
```

Nastavenia potrebuje len predná textúra a dočasná zadná textúra, do ktorých sa budú ukladať výsledky. Hĺbková textúra ukladá iba momentálnu najbližšiu a najvzdialenejšiu "z" súradnicu.

```
glDrawBuffer(drawBuffers[0]);
glClearColor(-MAX_DEPTH, -MAX_DEPTH, 0, 0);
glClear(GL_COLOR_BUFFER_BIT);
glBlendEquation(GL_MAX);
DrawScene(MVP, initShader);
```

Prvá funkcia nastaví zobrazovací cieľ, do ktorého sa bude vykresľovať. Je naviazaný na hĺbkovú textúru. Inicializujeme ju dvoma hodnotami najnižšej hĺbky, a keďže uchováva len dve hodnoty s pohyblivou desatinnou čiarkou, tak posledné dve sa nastavia na nulu. Ako funkcia miešania sa nastaví "GL_MAX", ktorá vráti najväčšiu hodnotu z farieb. Pri vykreslení sa zavolá VAO s požadovaným objektom, ktorý je potrebné vykresliť a pošle sa pre spracovanie do shadru. V shadri sa vykoná nasledovná akcia:

```
vFragColor.xy = vec2(-gl_FragCoord.z, gl_FragCoord.z);
```

Ako výstup do hĺbkovej textúry, zapíše najvzdialenejší a najbližší fragment.

Pre textúru, ktorá uchováva zadnú farbu, je inicializácia nasledovná.

```
glDrawBuffer(drawBuffers[6]);
glClearColor(bg.x, bg.y, bg.z, 0);
glClear(GL_COLOR_BUFFER_BIT);
```

Vyčistí sa výslednou farbou pozadia, ktorú chceme zobraziť. Alfa kanál pozadia sa nastavuje podľa toho, ako je žiadané, aby zadná farba ovplyvňovala výsledok. Pokiaľ je alfa kanál nastavený na hodnotu nula, vôbec sa s ním neráta.

Táto časť sa bude vykonávať, pokým algoritmus neodoberie všetky vrstvy. Koniec cyklu bude určovať buď počet vrstiev, ktoré sa vypočítajú zo zadanej hodnoty počtu priechodov vynásobených dvoma a od výsledku ešte odrátaná jednotka, alebo pomocou "occlusion query", ktorý zisťuje, či sú viditeľné časti objektu.

```
int numLayers = (NUM_PASSES - 1) * 2;
for (int layer = 1; bUseOQ || layer < numLayers; layer++)
```

V tomto cykle sa bude odohrávať odoberanie vrstiev. Najprv po každej iterácii je potrebné vyčistiť prednú aj zadnú dočasnú textúru, do ktorých sa budú zobrazovať výsledky z fragment shadru. Zaistíme, že nenastanú žiadne hazardy prepisovania. Farba sa inicializuje tak, aby neovplyvňovala miešanie farieb, takže alfa kanál sa nastaví na hodnotu nula.

```
glDrawBuffers(2, &drawBuffers[bufId+1]);
glClearColor(0, 0, 0, 0);
glClear(GL_COLOR_BUFFER_BIT);
```

Opísaný spôsob volania funkcie "glDrawBuffers" zaistí, že nastaví ukazovateľ na "bufId+1" a pri ďalšej hodnote poľa sa tento ukazovateľ posunie o jednu pozíciu. Premenná "bufId" je nastavená na momentálnu vrstvu, ktorá je vydelená dvomi a jeho zvyšok je aktuálna pozícia pre zobrazovacie ciele. Tým sa zaistí, že sa textúry budú stále striedať.

```
currId = layer % 2;
int prevId = 1 - currId;
int bufId = currId * 3;
```

Pre ukladanie nových hodnôt do hĺbkovej textúry ju je potrebné znova inicializovať, aby nenastalo zlé nastavenie hodnôt.

```
glDrawBuffer(drawBuffers[bufId+0]);
glClearColor(-MAX_DEPTH, -MAX_DEPTH, 0, 0);
glClear(GL_COLOR_BUFFER_BIT);
```

Uvedeným súborom kódu sa zabezpečí, že nenastane žiadne nesprávne prepísanie všetkých textúr, ktoré budeme používať.

Textúry, do ktorých sa bude zapisovať v nasledujúcom kroku sú tri. Jedna na prednú, jedna pre dočasnú zadnú a jedna pre hĺbkovú textúru.Pre všetky sa bude používať "MAX blending". Preto nie je možné hneď použiť aj zadnú textúru.

```
glDrawBuffers(3, &drawBuffers[bufId+0]);
glBlendEquation(GL_MAX);
```

Pre zmiešavanie je potrebné, aby sa predošlé výsledky zobrazenia poslali do shadru a zmixovali sa s aktuálnymi vrstvami.

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_RECTANGLE, depthTexID[prevId]);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_RECTANGLE, texID[prevId]);
```

Po nastavení týchto textúr a poslaní do shadru, ich spracuje. Dočasná zadná textúra sa nebude nastavovať, pretože s ňou sa nebude pracovať.

Pre korektné zobrazenie objektov, ktoré vyžadujú určitý druh tieňovania sa bude používať jednoduchá metóda zistenia polohy vrcholu, pozície kamery a smeru jeho normálu. Normál určuje, ktorým smerom vrchol odráža svetlo. Je to inverzný vektor voči skalárnemu vektoru.

```
float diffuse = abs(normalize(NormalMatrix*vNormal));
color.rgb *= diffuse;
```

Predchádzajúcou časťou kódu sa nenaimplementuje osvetlenie, ale iba ilúzia tieňu. Uvedený spôsob sa využíva v dôsledku jeho jednoduchosti.

Pre každý vrchol je potrebné určiť, kde sa na výstupe bude nachádzať, aby nedošlo ku nekorektnému spracovaniu obrazu. Táto operácia sa bude vykonávať u každého vertex shadru.

Spracovanie jednotlivých vrstiev sa bude odohrávať vo fragment shaderi.

```
float fragDepth = gl_FragCoord.z;
vec2 depthBlender = texture(depthBlenderTex, gl_FragCoord.xy).xy;
vec4 forwardTemp = texture(frontBlenderTex, gl_FragCoord.xy);
vFragColor0.xy = depthBlender;
vFragColor1 = forwardTemp;
vFragColor2 = vec4(0.0);
float nearestDepth = -depthBlender.x;
float farthestDepth = depthBlender.y;
```

V prvom kroku získa najbližšiu hĺbkovú hodnotu, aby sa zistilo, ktorú vrstvu bude odoberať. Následne nadobudne hĺbkovú textúru, z ktorej je potrebné získať len súradnice "x" a "y", pretože uchováva iba momentálnu maximálnu a minimálnu hĺbku predošlej vrstvy. Predná textúra uchováva prednú farbu. Táto hodnota bude uložená v premennej "forwardTemp". Premenná "vFragColor0" bude zapisovať do hĺbkovej textúry, a následkom toho sa jej priradí hodnota predošlej. Premenná "vFragColor1" bude zapisovať vo forme výstupu do prednej textúry. Premenná "vFragColor2" bude mať ako výstup dočasnú zadnú textúru. Vynulujeme ju pre budúce zapisovanie aby nenastalo nežiaduce zmiešanie "MAX_BLEND" funkciou.

Na stanovenie viac alebo menej viditeľného fragmentu, podľa toho aká je hodnota alfa, je potrebné získať inverzný alfa kanál. Pokiaľ bude nízka hodnota alfa kanálu, fragment za ním by mal byť tiež viditeľný a ich farba vo výsledku bude správna.

```
float alphaMultiplier = 1.0 - forwardTemp.w;
```

Zistenie, či odoberáme správnu vrstvu závisí od otestovania, či vrstva, ktorá sa bude spracovávať, je najbližšia alebo najďalej vzdialená. Pokiaľ je fragment vo väčšej vzdialenosti ako je najvzdialenejší fragment, alebo je bližšie ako najbližší nespracovaný fragment, tak musíme zaistiť aby sa už nespracovával. Zaistíme to tak, že sa mu nastaví najväčšia možná hĺbka, ktorú algoritmus akceptuje.

```
if (fragDepth < nearestDepth || fragDepth > farthestDepth) {
            vFragColor0.xy = vec2(-MAX_DEPTH);
            return;
}
```

Pokiaľ je vzdialenosť fragmentu v rozmedzí ešte dvoch nespracovaných, bude sa ešte spracovávať, preto ho preskočíme.

```
if (fragDepth > nearestDepth && fragDepth < farthestDepth) {
      vFragColor0.xy = vec2(-fragDepth, fragDepth);
      return;
}
```

Pokiaľ fragment prešiel týmito testami, je potrebné ho spracovať. Pre zaistenie, že už sa nebude spracovávať, je potrebné mu nastaviť maximálnu hĺbku. Ďalej je potrebné navoliť farbu fragmentu. V tomto prípade sa vyberá, či má byť fragment osvietený alebo nie.

```
vFragColor0.xy = vec2(-MAX_DEPTH);
vec4 Color;
if(isObject == 0.0)
        Color = vColor;
```

```
else
        Color = Lighted();
```

Pokiaľ sa bude hodnota hĺbky rovnať najbližšej ešte neodobranej, tak to znamená, že vrstva je najbližšie a bude sa ukladať do prednej textúry. Keďže GLSL povoľuje iba jednu funkciu na miešanie, je potrebné urobiť predchádzajúci krok ručne. Nie je možné použiť zabudovanú funkciu. Po pridaní výstupu výslednej farby do "vFragColor1" sa vyberie farba z väčšou hodnotou z predošlej prednej textúry a výstupu.

```
if (fragDepth == nearestDepth) {
      vFragColor1.rgb += Color.rgb * Color.a * alphaMultiplier;
      vFragColor1.w = 1.0 - alphaMultiplier * (1.0 - Color.a);
```

V prípade, že to je zadná vrstva, tak sa uloží a pošle sa na ďalšie spracovanie.

```
} else {
      vFragColor2 += Color;
}
```

Pre zmiešavanie zadnej textúry použijeme odlišný vzorec ako "GL_MAX". Je to preto, že v tomto smere sa farba nemusí navršovať.

```
glDrawBuffer(drawBuffers[6]);
glBlendEquation(GL_FUNC_ADD);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Zistenie, či je vôbec možné nejaké body do vykresľovacieho zásobníka vykresliť, zabezpečíme použitím "occlusion query". Ten zaistí, že pokiaľ fragment shader nevykreslí žiadny výstup, tak sa odoberanie vrstiev ukončí. Preto sa musí nastaviť začiatok testovania.

```
if (bUseOQ) {
      glBeginQuery(GL_SAMPLES_PASSED_ARB, queryId);
}
```

Samotné miešanie zadnej textúry a dočasnej zadnej textúry sa vykoná v ďalšom GLSL súbore. Je dôležité, aby štvorec, na ktorý sa bude vykresľovať, bol zarovnaný na výšku a šírku obrazovky, pretože môže nastať nekorektné zmiešanie textúr. Príčinou tohto je farba, ktorá môže ostať a bola nastavená na začiatku pred spustením algoritmu a taktiež fragmenty môžu byť posunuté o pár jednotiek od prednej textúry.

```
vFragColor = texture(tempTexture, gl_FragCoord.xy);
if(vFragColor.a == 0)
      discard;
```

Nastaví sa výstup farby a o zmiešanie farieb sa postará OpenGL. Odpadáva povinnosť písať vlastný vzorec pre zmiešanie farieb ako tomu bolo pri prednej textúre. Fragmenty, ktoré sú stopercentne priehľadné, neberieme do úvahy, keďže vôbec neovplyvnia výsledok.

Po ukončení miešania otestujeme, či bol zapísaný nejaký výsledok a pokiaľ nie, tak ukončíme odoberanie.

```
if (bUseOQ) {
    glEndQuery(GL_SAMPLES_PASSED);
    GLuint sample_count;
    glGetQueryObjectuiv(queryId, GL_QUERY_RESULT, &sample_count);
    if (sample_count == 0) {
          break;
     }
  }
}
```

Po ukončení odoberania vrstiev, už nie je viac potrebné mať zapnuté miešanie farieb. Zmiešanie zadnej a prednej textúry bude zaistené ručne. Taktiež, vo finálnej verzii, sa bude vykresľovať na obrazovku, preto je potrebné odstrániť všetky naviazané zásobníky fragmentov. Zo zásobníka sa bude vyberať iba zadná ľavá časť zásobníku. Vo vertex shaderi sme zabezpečili, že sa fragmenty budú zapisovať len do zadného ľavého zásobníka, tak sa bude získavať obraz len z tejto lokácie

```
glDisable(GL_BLEND);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glDrawBuffer(GL_BACK_LEFT);
```

Pre výsledné zmiešanie prednej a zadnej textúry sa použije nasledujúci GLSL algoritmus:

```
vec4 frontColor = texture(frontBlenderTex, gl_FragCoord.xy);
vec3 backColor = texture(backBlenderTex, gl_FragCoord.xy).rgb;
float alphaMultiplier = 1.0 - frontColor.w;
vFragColor.rgb = frontColor.rgb + backColor * alphaMultiplier;
```

Je potrebné nastaviť alfa multiplikátor pre zadnú textúru, aby boli pre obe textúry zjednotené. Získava sa zo zadnej textúry a touto hodnotou sa vynásobia všetky farby fragmentu. Alfa kanál sa odčítava preto, aby sa získala inverzná hodnota. Takže pokiaľ bude veľká hodnota alfa kanálu prednej textúry, tak predné farby budú viac viditeľné a naopak. Keď je zaistené, že majú rovnakú alfu, tieto farby sa sčítajú.

Po týchto úkonoch je celý algoritmus dokončený a na obrazovku sa vykreslí výsledný obraz.

6. Záver

Cieľom bakalárskej práce je stručný prehľad správneho zobrazovania priehľadných plôch nezávisle na poradí. Bol poskytnutý pohľad na tieto techniky a spôsob ich implementácie s referenciami na zdroje, ktoré sa týmto zaoberajú podrobnejšie. Hlavným cieľom však bolo naimplementovať jednu zo spomenutých techník. Bol vybraný dual depth peeling.
Kapitola 3 je rozobratá z matematického hľadiska, teda ako sa tieto spôsoby odohrávajú vo výpočtoch u všetkých metód. Ďalej tu bol popísaný spôsob, ako sa má inicializovať a spracovávať v teoretickej rovine. Výsledná implementácia dual depth peelingu je popísaná v kapitole 5.

Výsledkom práce je funkčné zobrazenie pomocou dual depth peelingu, ktorý je nezávislý na poradí vrcholov. Pre ukážku boli vybrané kocky a drak, vymodelovaný univerzitou Stanford. Na týchto objektoch som demonštroval uvedenú techniku. Pri testovaní tejto metódy s phongovým osvetľovacím modelom sa mi implementácia nepodarila, čo je jedným z mojich cieľov do budúcna. Pri vykresľovaní bolo zistené, že táto metóda je náročná na výkon a spomaľuje sa tým, čím viac polygónov objekt obsahuje. Ďalej bolo zistené, že pri výslednej farbe vypočítanej touto metódou je objekt sýtejší, a to následkom daného sčítavania farieb (príloha B. I. a príloha B. II.). Pri modeli draka je taktiež výsledok o niečo sýtejší, ale so správnym zmiešaním farieb. Je teda možné usúdiť, že táto metóda je vhodná pre scény, ktoré sú predvypočítané, napríklad použitie v animovaných filmoch.

Literatúra

[1.] Bavoil, L., Myers, K. Order Independent Transparency with Dual Depth Peeling. 2008. [2.] STEINBERGER, Ralf, Bruno POULIQUEN a Johan HAGMAN. Cross-Lingual Document Similarity Calculation Using the Multilingual Thesaurus EUROVOC. [online]. 2002, s. 10 [cit. 2012-05-03]. DOI: 10.1007/3-540-45715-1_44. Dostupné z: http://www.springerlink.com/content/d0q93wxbpkaqya85/fulltext.pdf

[2.] RIGAZZI,Alessandro. *Http://www.slideshare.net/acbess/order-independent-transparency-presentation*. Swiss National Supercomputing Centre, 2008. Dostupné z: http://www.slideshare.net/acbess/order-independent-transparency-presentation.http://www.eng.utah.edu/~cs5610/lectures/OrderIndependentTransparency.pdf

[3.] EVERITT, Cass. *Order-Independent Transparency*. NVIDIA Corporation, 2009. Dostupné z: http://www.eng.utah.edu/~cs5610/lectures/OrderIndependentTransparency.pdf

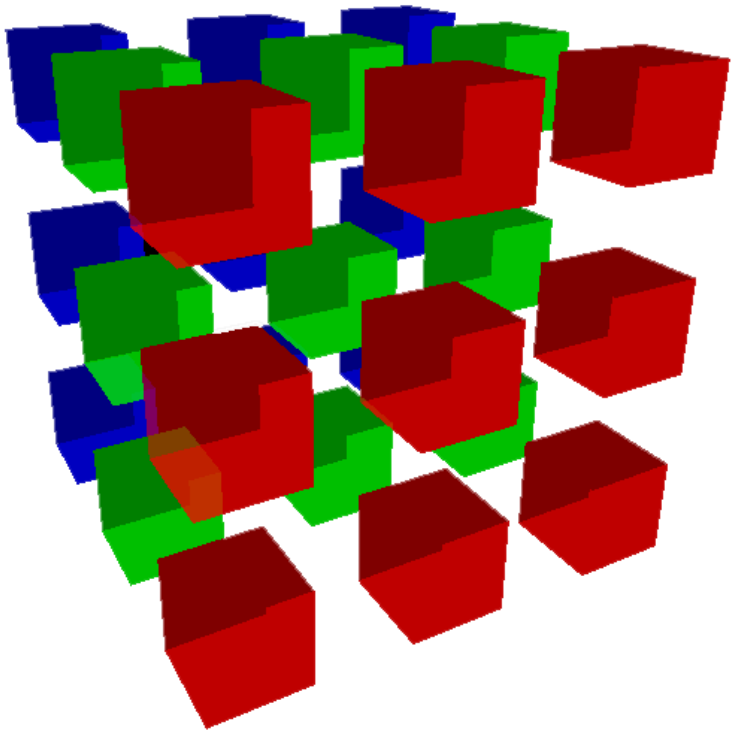
Prílohy

A. Obsah priloženého DVD

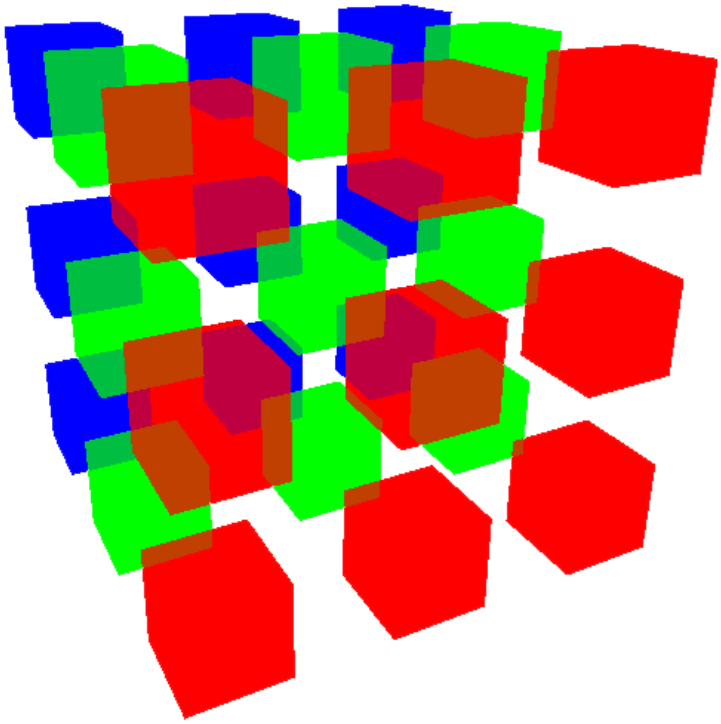
Adresár	Obsah adresára
/app/	Zdrojové kódy komponenty a jej implementácie
/text/	Bakalárska práca v elektronickej podobe

B. Výsledné obrázky

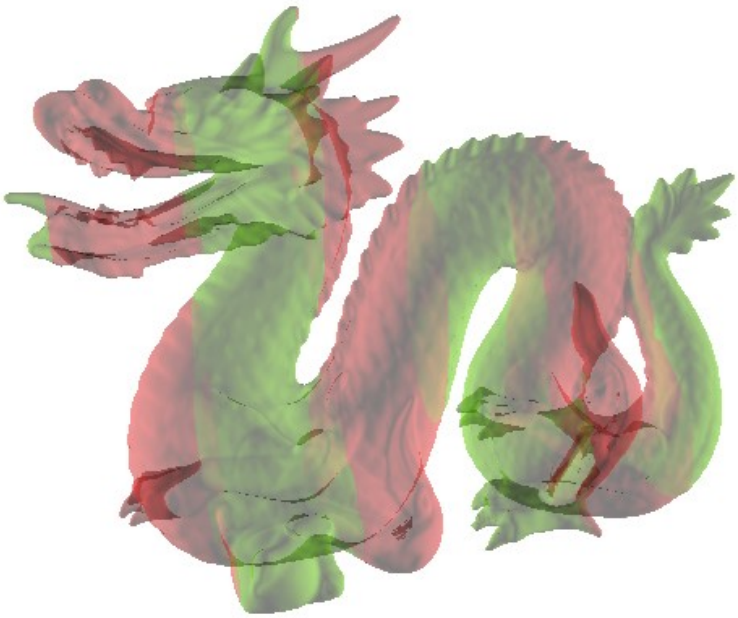
I. Zobrazenie priehľadných objektov pri metóde zabudovanej v OpenGL bez tieňovania



II. Zobrazenie pri zobrazení algoritmom dual depth peeling bez tieňovania



III. Zobrazenie priehľadného objektu pri metóde zabudovanej v OpenGL s tieňovaním



IV. Zobrazenie priehľadného objektu s metódou dual depth peelingom pri tieňovaní